VIA PadLock
Random Number Generator
Programming Guide

Version 1.32

This is **Version 1.32** of the VIA PadLock Random Number Generator Programming Guide.

LIFE SUPPORT POLICY

VIA processor products are not authorized for use as components in life support or other medical devices or systems (hereinafter called life support devices) unless a specific written agreement pertaining to such intended use is executed between the manufacturer and an officer of VIA.

1. Life support devices are devices which (a) are intended for surgical implant into the body or (b) support or sustain life and whose failure to perform, when properly used in accordance with instructions for use provided in the labeling, can be reasonably expected to result in a significant injury to the user.
2. This policy covers any component of a life support device or system whose failure to perform can cause the failure of the life support device or system, or to affect its safety or effectiveness.

Table of Contents

CHAPTER

1

INTRODUCTION

VIA provides a suite of security technologies called Padlock in all new processors. The first Padlock technology, PadLock Random Number Generator (RNG), introduced in the VIA Nehemiah processor, provided a fast hardware random number generator *on the processor die*.

The VIA Nehemiah processor (stepping 8) extended this feature by placing two separate RNG noise sources on the die, improving both performance and randomness.

The VIA Esther processor has the same RNG capabilities as the VIA Nehemiah stepping 8 processor. While the functionality is the same, the Esther and Nehemiah processors are manufactured with different silicon technologies, and there may be subtle differences in the entropy and statistical properties of the random bits produced by these processors.

Background information on the theory of random numbers, their use in computing, and details regarding Centaur Technologies testing methodologies and the statistical properties of random number produced by the PadLock RNG may be found in the PadLock Security Application Note.

The CPUID identification of the VIA Nehemiah processor is:

Vendor ID: CentaurHauls
Family: 6
Model: 9
Stepping: 3 (or higher)

The CPUID identification of the VIA Esther processor is:

Vendor ID: CentaurHauls
Family: 6
Model: 10
Stepping: (all)

It is theoretically possible that even with a correct stepping, PadLock RNG may not be present. Thus one must still check for existence of PadLock RNG as described in section 2.2.

For support, interested developers should contact: rng_support@centtech.com.

CHAPTER

2

PROGRAMMING INTERFACE

2.1 ARCHITECTURE

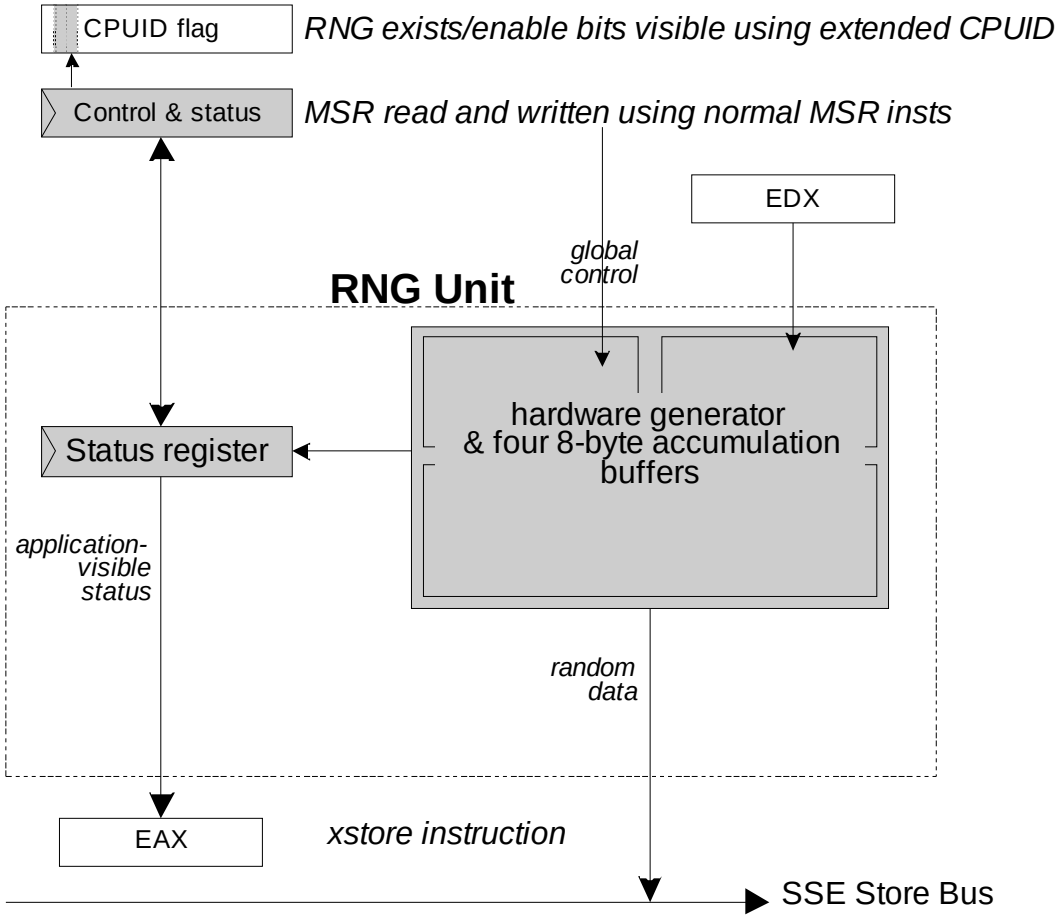
Figure 1 is a conceptual view of the VIA PadLock Random Number Generator (RNG). The major components are:

- Two bits in the Centaur extended feature flags returned by the CPUID instruction. These bits identify (1) whether PadLock RNG physically exists on the processor and (2) whether it is currently enabled.
- An RNG status and control Machine Specific Register (MSR). In addition to providing status information, this privileged register controls global options of PadLock RNG.
- A high-speed hardware mechanism for generating random bits.
- A hardware datapath mechanism that collects the generated bits into multiple buffers.
- A non-privileged x86 instruction - XSTORE - that stores the collected random bits to memory, using the SSE store bus, as well as placing a copy of the status and control MSR in EAX. An application can then verify that PadLock RNG is configured acceptably.
- An option to store only a specified subset of the generated bits, reducing the effective bit generation rate but improving the statistical randomness of the data.
- A REP version of the XSTORE instruction, which allows easy collection of long contiguous strings of random data while allowing transparent interrupts.

Subsequent sections in this chapter describe in detail these registers and instructions. The basic approach to using VIA PadLock RNG is simple:

1. If special configuration is required (e.g., for testing purposes), set control values via the RNG MSR. This is a privileged operation and rarely needs to be done.

Figure 1. Visible RNG Architecture



2. An application can use XSTORE at any time to store both the random data *and* the instantaneously accurate count of how many random bytes have been stored. This is an atomic instruction and can be used concurrently by any number of active tasks (and the operating system). Note, however, that any global options set in step 1 affect all users.

3. Examine the number of bytes returned in step 2, and repeat step 2, if necessary, until the desired number of bytes has been obtained. Or ...

4. Replace steps 2 and 3 with a REP XSTORE instruction specifying the exact number of bytes desired.

Paranoid (e.g., cryptographic) applications are advised to verify that the RNG configuration is valid (or at least unchanged) after each XSTORE operation. The concern is that a different application running on the CPU could modify the RNG configuration. Centaur Technology recommends that paranoid applications not use the REP prefix.

Alternately, we recommend that paranoid applications configure PadLock RNG for the worst entropy level with the whitener OFF and the EDX divider equal to zero, and mix these raw bits (along with anything else in the entropy pool) with a good mixing function such as the Secure Hash Algorithm (SHA-1). The worst that a hostile application can do is slow or disable the RNG. But our slowest speed (whitener ON, EDX divider at the maximum, only one noise source sampled, VIA Nehemiah processor, slowest DC Bias setting) is still 800K bits/second.

The VIA Esther processor includes PadLock Hash Engine (PHE), a hardware implementation of both SHA-1 and the newer SHA-256. Please see the VIA PadLock Hash Engine Programming Guide for further details.

2.2 CONFIGURING AND ENABLING

To use VIA PadLock RNG, three conditions must be met:

5. PadLock RNG must physically exist on the chip. Its existence can be discovered from the Centaur Extended CPUID Feature Flags. This condition is set as part of the manufacturing process and cannot be changed by software.

6. PadLock RNG must be enabled and configured appropriately. The enabled state can be discovered from the Centaur Extended CPUID Feature Flags. PadLock RNG can be enabled or disabled by writing to an MSR. The RESET default is enabled. But, see the next condition.

7. SSE instructions must be enabled via the standard x86 method of enabling the FXSAVE/FXRSTOR instructions using CR4[9] This enables the full set of SSE instructions. If CR4[9] is not set, PadLock RNG behaves as if it were disabled via the RNG MSR, regardless of the setting of the RNG enable bit in the RNG MSR.

Note: This dependency on SSE is because the RNG datapath just happens to be buried inside the SSE datapath. This is an implementation choice and future VIA processors may eliminate this dependency.

2.2.1 CPUID INSTRUCTION IDENTIFICATION

The newer VIA processors (both VIA Nehemiah and VIA Esther) support the Centaur Extended Feature Flags. When the CPUID instruction is executed with EAX = 0xC0000000, the processor returns 0xC0000001 in EAX.

If a CPUID with EAX = 0xC0000000 returns a value in EAX >= 0xC0000001 then Centaur Extended Feature Flags are supported. A CPUID with EAX = 0xC0000001 then returns the Centaur Extended Feature Flags in EDX. There are two bits in EDX that describe PadLock RNG:

- **EDX[2]** If this bit = 0, PadLock RNG does not exist on this chip. A RDMSR/WRMSR of MSR 0x110B will cause a General Protection Fault, and the XSTORE instruction will always cause an Invalid Opcode Fault. If this bit = 1, PadLock RNG exists, and RDMSR/WRMSR of MSR 0x110B is allowed. The behavior of XSTORE is dependent upon whether PadLock RNG is enabled or not.
- **EDX[3]** If this bit = 0, PadLock RNG is disabled. The XSTORE instruction will cause an Invalid Opcode Fault. This bit reflects the result of setting bit 6 of MSR 0x110B. If this bit = 1, PadLock RNG is enabled (usually the RESET default), and the XSTORE instruction can be used at any privilege level (provided that CR4[9] =1)

2.2.2 MSR 0X110B

MSR 0x110B provides both status and control of PadLock RNG. In addition, a copy of the MSR contents as they are at the start of execution of an XSTORE instruction is placed in EAX as a result of executing an XSTORE instruction.

The EDX portion of this MSR is undefined. The EAX portion of the MSR contains (for VIA **Esther** processors):

31:14	13:13	12:10	9:8	**7:7**	6:6	5:5	4:0

Reserved	Raw bits enable	DC bias	Noise Source select	*Reserved*	RNG enable	*Reserved*	Current byte count

On VIA **Nehemiah** processors, there are some additional bit fields defined:

31:22	21:16	15:15	14:14	13:13	12:10	9:8	**7:7**	6:6	5:5	4:0
Reserved	String filter count	String filter failed	String filter enable	Raw bits enable	DC bias	Noise Source select	*Reserved*	RNG enable	*Reserved*	Current byte count

Reserved bits have undefined and unpredictable values. In the following detailed description, **(O)** means a read-only field output by the processor, and **(I)** means the field can be written by software. The phrase "when the MSR is read" means when it is read by a RDMSR instruction or when it is copied into EAX as part of an XSTORE instruction execution.

VIA recommends that when writing the PadLock RNG MSR, all of the reserved bits should be set to zero. It is not possible to improperly configure current versions of the VIA PadLock RNG, but subsequent versions may define these reserved bits and software that incorrectly detects the processor stepping may improperly configure these later steppings.

Any write to MSR 0x110B causes a *load reset* of the PadLock RNG hardware. The PadLock RNG actions on load reset are:

The internal hardware buffers are cleared to zero.

The various options defined by the MSR are set to the value written to the MSR.

The detailed fields are:

Bits 4:0 Current Random Byte Count (O): The meaning of this field is different for VIA Nehemiah and VIA Esther processors.

For VIA **Esther**:

When the MSR is read, this *read-only* field reports the number of eight-byte accumulation buffers currently full. Each bit in the field represents a single buffer. Thus:

0x00 no accumulation buffers available

0x01 one accumulation buffer available

0x03 two accumulation buffers available

0x07 three accumulation buffers available

0x0F four accumulation buffers available

Other values in this field when reported from a MSR read operation are not possible.

When this field appears in the copy of the MSR placed in EAX by the execution of XSTORE, the number is the *exact number of bytes actually stored*.

For VIA **Nehemiah**:

When the MSR is read, this *read-only* field contains the exact number of random bytes that are *currently* available for storing. Due to the asynchronous generation of random bytes, and the fact that multiple

programs may be storing data using XSTORE, the number that appears may not be the same as the number stored on the next execution of XSTORE.

When this field appears in the copy of the MSR placed in EAX by the execution of XSTORE, the number is the *exact number of bytes actually stored*.

Bit 6 RNG Enable (I): When set to 1, this bit enables PadLock RNG. When set to 0, the XSTORE instruction becomes invalid and the random number generator is internally disabled to reduce power consumption (see section 4.3). The extended CPUID feature flag that indicates whether or not PadLock RNG is enabled (EDX[3]) is a copy of this bit.

This enable bit may be set internally by the RESET process. In this case, the effect on the processor is the same as if a write MSR set the enable bit.

Bits 9:8 Noise Source Select (I): These bits control the two noise sources on the processor that input bits to the accumulation buffers.

00 Noise source "A" active

01 Noise source "B" active

1x Both noise sources active

On VIA **Nehemiah** processors prior to stepping 8, these bits are reserved and undefined. The default RESET state is for both bits to be zero.

Bits 12:10 DC Bias (I): These bits control the DC bias supplied to the random number generator (see chapter 3). These bits may affect the speed of the generator and the randomness of the generated bits. There is no necessary correlation between the way these bits affect VIA Esther and VIA Nehemiah processors. Users who like to experiment with their RNG are advised not to expect that settings appropriate for the VIA Nehemiah processor will provide comparable behavior on the VIA Esther processor.

Bit 13 Raw Bits Enabled (I): If this bit is set to 1, the von Neumann compressor (or *whitener*) in the hardware generator is *disabled* and the raw bits produced by the random generator are delivered to the accumulation buffers (see the hardware summary in chapter 3). A 0 in this bit selects the whitener function.

Bit 14 String Filter Enable (I): For VIA **Nehemiah** processors only. If set to 1, this enables a test feature that filters out strings of contiguous ones or zeroes longer than the value set in bits 21:16. Note that enabling the string filter may introduce non-uniform statistical characteristics in the output.

Bit 15 String Filter Fail (I/O): For VIA **Nehemiah** processors only. This bit indicates that while the string filter was enabled, the hardware detected a string of contiguous ones or zeroes longer than the value defined by the filter count in bits 21:16. Only the hardware can set this bit to a 1, but a MSR write can set it to 0**.**

Bits 21:16 String Filter Count (I): For VIA **Nehemiah** processors only. The value in this field defines the bit-string length (8 - 63) to be used by the string filter. *Bit string lengths less than 8 will produce unexpected results and should not be used*.

2.3 x86 XSTORE INSTRUCTION

VIA PadLock RNG implements the x86 instruction: XSTORE. This instruction is enabled by the RNG enable bit in MSR 0x110B. An Invalid Instruction exception occurs if XSTORE is executed when not enabled.

2.3.1 XSTORE: STORE RANDOM DATA INSTRUCTION

The XSTORE instruction stores zero, one, two, four, or eight random bytes in memory and places a copy of the current value of the RNG MSR 0x110B in EAX. This copy is referred to as the *status word*.

The opcode of XSTORE is an invalid opcode in other x86 processors. When enabled, the format of the XSTORE instruction is:

0x0F	0xA7	0xC0

The XSTORE instruction requires that:

ES:EDI points to the starting (low-order) memory address where the random data are to be stored. No segment override is possible.

EDX specifies the rate at which the random data are returned, and the corresponding level of randomness. Only the lower two bits are meaningful, and the upper 30 bits will be set to zero by the instruction.

The XSTORE instruction has many similarities with the x86 STOS instruction:

The value in EDI is updated for the number of bytes stored.

An address-size prefix affects the size of EDI used and updated.

A REP prefix affects the behavior of the XSTORE instruction.

The usual address exceptions (past the segment limit, etc.) can occur.

And many contrasts:

The REPNE prefix is undefined.

In addition to storing the random bytes, the status word is placed in EAX.

An operand size prefix causes an Invalid Instruction exception.

The specification of a 16-bit mode in CS is ignored; the full 32-bit EAX is always updated.

The DF (direction flag) in EFLAGS has no effect: instruction execution always adds the number of random bytes stored to EDI.

Either zero, one, two, four or eight bytes are stored to memory depending on the data rate specified by EDX and the status of the PadLock RNG hardware. Software must always provide an appropriately sized write-enabled buffer at the ES:EDI destination address.

Zero, one, or two 4-byte stores perform the store. Thus, all memory-operand restrictions and exceptions that are applicable to a four-byte store are also applicable to the XSTORE instruction. If alignment check is enabled and is to be avoided, then the address in the starting ES:EDI should be 4-byte aligned. Similarly, segment limit checks, protection check, page faults, etc. behave as if the data size is four bytes.

2.3.2 XSTORE: NUMBER OF BYTES STORED

The rules for the number of bytes stored by an XSTORE instruction:

- The instruction tests whether a hardware accumulation buffer is full. If fewer than 8 bytes are available, the instruction performs no store, and returns with a zero byte count in the status word.

- If at least 8 bytes are available in the hardware buffers, the appropriate bits are selected (defined by EDX, more on this subsequently) and either one or two four-byte stores are performed as required.
- Note there may be less than four valid *random* bytes stored depending on EDX. In this case, the remaining bytes of the store contain zeroes. The byte count in EAX and the update to EDI are based on the number of random bytes stored, not on the size of the stores themselves.

Here is a summary:

EDX 1:0 value	Total bytes available	Total bytes stored	Random bytes stored	EAX[4:0] & EDI update byte count
N/A	< 8	0	0	0
00	>= 8	8	8	8
01	>= 8	4	4	4
10	>= 8	4	2	2
11	>= 8	4	1	1

2.3.3 REP XSTORE INSTRUCTION

The REP prefix on an XSTORE instruction performs as follows:

- The ECX value defines the total number of random bytes to be stored.
- The XSTORE instruction repeats until the requested number (in ECX) of random bytes is stored.
- Zero, one or two 4-byte stores are performed for each repeat of XSTORE. ECX and EDI are updated automatically such that random bytes are concatenated into a contiguous string and processing continues until the desired quantity of *random* bytes is stored.

The implication of storing fewer random bytes than the size of the store is that unaligned four-byte stores may be performed. These are slower than aligned stores and will cause an alignment check if it is enabled. However, the asynchronous nature of the random number generator means that any alignment checks taken during a REP XSTORE will overlap with the generation of new bits and there will be no real-world performance penalty.

Another implication is that on the last iteration, REP XSTORE may store up to seven bytes to memory beyond the last random byte requested. Adequate memory must be allocated based on the EDX value to contain the last one or two four-byte stores.

The REP XSTORE execution is interruptible. When an exception or interrupt occurs and is taken, or the instruction completes execution:

1. ES:EDI is updated to address the next location for storing *random* byes
2. ECX is updated to contain the number of random bytes remaining to be stored, and
3. EAX contains a copy of the current status word. *However, the byte count field (4:0) in the status word is not defined for the REP case.*

Code that runs during an interrupt could, in theory, modify the PadLock RNG configuration. In (unusual) environments where this may occur, application programmers should be aware that the RNG status reported in EAX at the end of the operation may not reflect the configuration at the beginning of the operation. If it is important to make sure that the RNG configuration does not change, XSTORE should be used, not REP XSTORE.

The same rules apply as for a non-REP XSTORE for the REPNE prefix, the AS prefix, the OS prefix, the size for alignment purposes, the size for other exceptions, etc.

Note: There is a theoretical problem with using the REP XSTORE instruction because (1) the instruction does not return control until all requested bytes are stored, and (2) the generation bit rate is variable. The REP version of XSTORE prevents easy detection of a situation where the RNG hardware is not returning any bytes. In practice, this not a real risk, since this situation cannot occur without a catastrophic hardware failure. Applications can address this possibility by performing a startup test.

2.4 STARTUP

During RESET, if PadLock RNG is present:

- MSR 0x110B is initialized with the value: 0x00000040. This enables PadLock RNG (bit 6 = 1) and sets the default values of all control fields to 0.
- The PadLock RNG hardware accumulation buffers are cleared and the accumulation count is set to zero.
- Random numbers start to accumulate overlapped with any other RESET or software function.

As opposed to RESET, the INIT signal causes no *direct* actions relative to PadLock RNG. That is, INIT does not internally cause a write to MSR 0x110B. The INIT signal causes CR4, however, to be reset. Since this resets the FXSAVE/FXRSTOR enable, CR4[9], PadLock RNG is indirectly disabled.

Start-up testing is a good idea in all environments. It is also a good idea to perform some test within an application before any collected random data is trusted.

Which startup test should be run is not obvious, however. There are two basic things that a startup test could be testing for:

- Whether PadLock RNG is basically working or not (for example, does it have a stuck bit such that no data is ever returned).
- The randomness characteristics of the numbers generated.

In normal operation (when the whitener is enabled), it is theoretically possible that the XSTORE instruction will generate no new bits for a long time. This theoretical delay has no guaranteed maximum, even in a correctly working part, in the same way that there is no limit to the number of sequential coin tosses that can come up heads. In practice, however, the likelihood of a long delay is negligible. This is based on (1) general statistical principles, (2) the inherent design of the random bit generator, and (3) our observations based on extensive testing. Our testing has never showed any deviation from the basic generation rate over long periods of continuous operation (many months). The bit rate seems to be nearly constant from startup to shutdown. If startup tests run successfully (i.e., the hardware has not failed), applications should not need to detect timeouts.

Of course, a very conservative application can easily add a time-out check, since the atomic XSTORE instruction is usually used in a loop counting the bytes stored. It is easy to add another count that can detect **N** XSTORE instructions with no data returned. Alternatively, a time limit can be set and checked by the application.

The timeout detection works because the XSTORE instruction returns to execute the next instruction even if no random data is available. The REP XSTORE form, however, does not return control until all of the random bytes requested have been stored. There is no built-in timeout check. If PadLock RNG has died, the instruction will wait forever. Since REP XSTORE is interruptible, however, this case hangs the application,

but not the operating system or other applications. It is also theoretically possible to detect a REP XSTORE timeout by intercepting an interrupt and examining the ECX count compared to some timer information about how long the REP XSTORE has been executing. This is very tricky and requires hooks into the operating system, so we recommend the following: do not use the REP XSTORE instruction if you are concerned about an RNG timeout condition.

2.5 MANUFACTURING TESTS

The VIA manufacturing tests includes a test of PadLock RNG to verify that the bit generators and the associated datapaths are alive and working with no stuck bits, etc. It does not assure that PadLock RNG will produce statistically random bits, but will generally detect catastrophic failures.

Important: *At this time*, if a processor fails the PadLock RNG test, or times out after a reasonable time without generating the expected number of bits, the random number generator on that part is permanently disabled. This means that PadLock RNG may be marked nonexistent on some Esther and Nehemiah processors. VIA may change PadLock RNG manufacturing tests at any time. Customers requiring PadLock RNG support may wish to contact VIA to ensure that all parts ordered have an active RNG.

2.6 DC BIAS

The DC bias adjustment is intended for Centaur Technology's internal analysis, test, and debug. However, our philosophy is to provide all existing functions to software in case there is some external value in them. Following that philosophy, we make the DC bias control available.

VIA PadLock RNG performs well with the default DC bias setting (0). It is possible, however, that alternate DC bias settings may improve bit-generation speed or randomness characteristics. The sophisticated or curious user may want to experiment with changing the DC bias. This requires protected mode access (for writing the PadLock RNG MSR).

One might think that a boot time characterization could be used to determine the optimal DC bias setting for a particular part. But an *extensive* number of samples must be collected and analyzed statistically, and the time required for a comprehensive determination of the correct DC bias is prohibitive for the boot process. Setting a non-standard DC bias may cause other applications to conclude that PadLock RNG is improperly configured.

For those users interested in optimizing the DC bias, Centaur Technology may be able to provide some advice or assistance based on our testing experience.

CHAPTER

3

HARDWARE DESCRIPTION

For ease in understanding, this discussion of the random bit generator describes the VIA Nehemiah processor (stepping 3) with a single set of oscillators. On VIA Esther processors, and on VIA Nehemiah Steppings 8 and higher, there are two independent hardware generators, both of which feed into the datapath logic.

Figure 2 provides a pictorial overview of PadLock RNG.

There are three fast oscillators, and one slow oscillator. These are free running ring oscillators; that is, their frequencies are not synchronized or stabilized and individually drift over time. The fast oscillator frequencies vary around a center of about 1 ns. The slow oscillator varies around a center of about 20 ns.

These oscillator frequencies are influenced by many factors:

Random quantum effects of the transistor behavior: thermal noise in the channel, gate leakage caused by tunneling, etc.

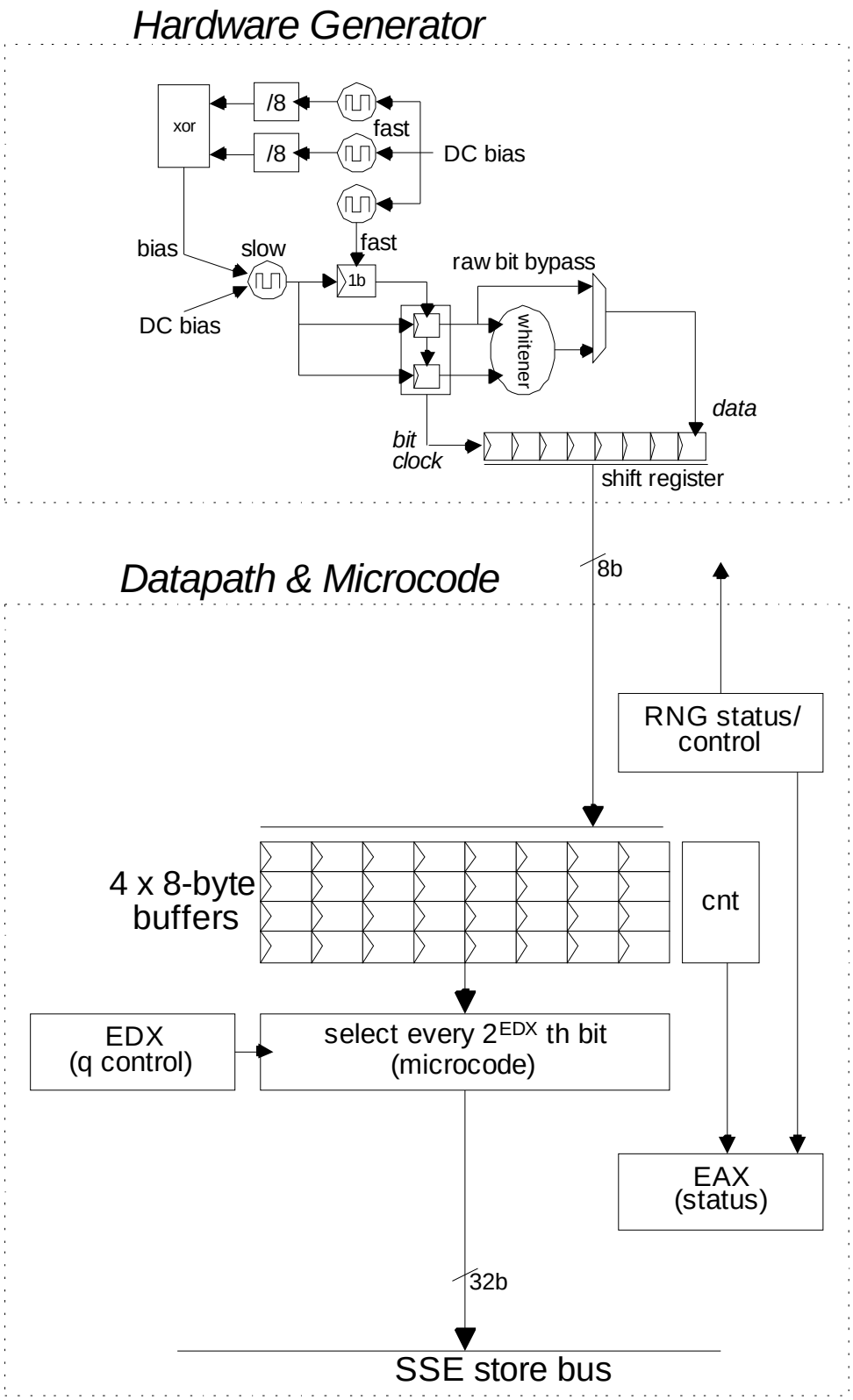
Dynamic voltage and temperature variations.

Dynamic electromagnetic effects.

Process differences in silicon dimensions. The oscillators are laid-out at right angles to each other to further accentuate silicon differences.

Process differences in silicon characteristics such as implant characteristics.

Figure 2. PadLock RNG Hardware Overview



The operating voltage of all oscillators is partially controlled by a single DC bias, which can be adjusted in eight steps via a programmable value.

The output of two of the fast oscillators is XOR'd to produce a variable signal. This variable signal further (in addition to the DC bias) modulates the voltage bias of the slow oscillator.

The output of the variably biased slow oscillator is used as the clock to sample a bit from the third free running fast oscillator.

Each two-bit pair generated is optionally processed through a von Neumann whitener, which may output a single bit or nothing. If a bit is produced, it is then accumulated in an eight-bit shift register. This whitener function will reduce or eliminate any residual bias in the hardware generator. For example, the whitener should correct for biases in the third oscillator's duty cycle.

The input/output function of the whitener is

Input	Output
00	Nothing
01	0
10	1
11	Nothing

When eight new bits have been accumulated, a strobe causes the eight bits to be grabbed by the RNG data path (which is running using the synchronous processor clock).

If both hardware noise generators have accumulated a byte during the same processor clock cycle, and both are selected, the datapath will accept only that byte from the "original" generator – the generator that corresponds to that existing on the first Nehemiah processors with RNG. The data from the second generator is lost. This is a rare event.

Once eight bits have been generated they are delivered to the synchronous datapath. The incoming bytes are placed sequentially in the next consecutive position of four 8-byte buffers. The hardware maintains FIFO pointers and a count so that it can properly store the incoming bytes and access the requested bytes.

The XSTORE instruction itself is implemented in microcode. When executed, the microcode checks to see if eight new bytes are available in one of the buffers. If not, it returns with a zero byte count indicated in the status word. When eight bytes are available, the microcode performs the one-of-N selection (specified by EDX), assembles the resulting data, and performs the store.

CHAPTER

4

PERFORMANCE

4.1 BIT GENERATION SPEED

The effective rate of generating random bits is variable and depends upon the processor's chip voltage, its temperature, process variations, phase of the moon, and so forth.

Note: sample size for the examples below is 10MB

For a VIA Nehemiah processor (stepping 8), with *both noise generators selected*, our data show that depending on temperatures and DC bias settings, bits are delivered to the accumulation buffers at about 80 Mb/s to 120 Mb/s.

With the whitener selected, the rate will fall to between 12 Mb/s and 20 Mb/s.

With only one noise source selected, or for stepping 3 of the VIA Nehemiah processor, the raw bit generation will be between 40 Mbs and 60 Mbs, and the whitened rates between 6 Mbs and 10 Mbs.

For a VIA Esther processor, with both noise generators selected, we have observed bit rates as high as 320 Mb/s and as low as 125 Mb/s, depending on the DC bias setting. At the default DC bias setting the bit rate should be about 200 Mb/s.

With the whitener selected, the rate falls to between about 32 Mb/s and 80 Mb/s.

Tests have also shown that there is typically a small correlation between sequential bits delivered to the accumulation buffers. To reduce this correlation, the XSTORE instruction uses the value in the EDX register (mod 4) as a divider, and returns only every 2^{EDX} bit from the accumulation buffers.

When both noise sources are active, bits accumulate in the buffers from both sources. As the noise sources are independent, there is some reduction in the bit-to-bit correlation but the effect is small. Use of both noise sources is recommended for faster generation of random bits, but should not be used to improve the entropy of the raw bits or the statistical qualities of the resulting random number stream.

4.2 INSTRUCTION TIMING

The XSTORE instruction is partly implemented in microcode. The number of execution clocks depends on

instruction does not return any random data unless an eight-byte accumulation buffer is full. If fewer than 8 bytes are available, the instruction returns status for the RNG with a zero byte count. When an accumulation buffer is ready, XSTORE returns the bits to memory as pointed to by ES:EDI and in the quantity specified by EDX. The status information in EAX correctly indicates the actual number of bytes stored.

The following table gives estimates for the number of clocks required for the XSTORE instruction. Note that a 2 GHZ Esther processor, with both of the noise sources active and with the whitener OFF, requires an average of about 500 clocks to fill a single 8-byte accumulation buffer.

Table 1. xstore Execution Clocks

EDX	Data Ready	Random Data Stored	Execution Clocks
0-3	No	0	≈30
0	Yes	8	≈55
1	Yes	4	≈310
2	Yes	2	≈200
3	Yes	1	≈120

REP XSTORE is executed as an internal microcode loop of individual XSTORE instructions until the specified number of bytes has been stored. If the requested count can be satisfied with a single internal XSTORE, and enough bits are in the hardware buffers, add 7 clocks to the values in Table 1.

For the more typical use of collecting a large number of bytes, the speed of REP XSTORE will be the bit generation speed plus a few hundred clocks.

4.3 POWER MANAGEMENT

PadLock RNG hardware is clock-enabled when the MSR enable bit is set. When the bit is set, the random number bit generator is enabled. In addition, the data path associated with accumulating bits, counting bytes, etc. is clock-enabled. Thus, dynamic power starts being consumed by PadLock RNG when it is enabled (normally by default at RESET).

Whenever all four 8-byte accumulation buffers are full, the random number bit generator clocks are temporarily disabled. When a buffer location becomes free, the generator is re-enabled. The associated RNG data path clocks are never disabled, however, until the MSR enable is cleared. Even though data path clocks are not disabled, the power consumption of this logic is very small.

Relative to the SSE unit power management, the XSTORE instruction behaves like SSE instructions. That is, when the XSTORE instruction is seen by the execution unit, the entire SSE unit is enabled until no further SSE instructions or XSTORE instructions are seen, at which point the SSE unit is disabled.

4.4 RANDOMNESS

Evaluation of the statistical qualities of the bits produced by VIA PadLock RNG is a complex process. Cryptography Research, Inc., of San Francisco has performed an independent evaluation. A copy of their report is available at: www.cryptography.com/resources/whitepapers/VIA_rng.pdf

A comprehensive discussion of randomness, of Centaur's internal testing methodology and results, and recommendations for using PadLock RNG in various environments is available in the *VIA PadLock Security Application Note*.

CHAPTER

5

SAMPLE CODE

```
/*
   Centaur RNG Analysis Program

   Copyright (C) 2003-2005 Centaur Technology, Inc.
   All rights reserved.

Redistribution and use in source and binary forms, with or without
modification, are permitted provided that the following conditions are met:

   * Redistributions of source code must retain the above copyright notice,
   this list of conditions and the following disclaimer.

   * Redistributions in binary form must reproduce the above copyright notice,
   this list of conditions and the following disclaimer in the documentation
   and/or other materials provided with the distribution.

   * Neither the name of Centaur Technology nor the names of its contributors
   may be used to endorse or promote products derived from this software
   without specific prior written permission.

THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS "AS IS"
AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE
IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE
ARE DISCLAIMED. IN NO EVENT SHALL THE COPYRIGHT OWNER OR  CONTRIBUTORS BE
LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR
CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF
SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS
INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN
CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE)
ARISING IN ANY WAY OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE
POSSIBILITY OF SUCH DAMAGE.

*/


#include <stdio.h>

#include <math.h>
```

```
#include <time.h>
#include <stdlib.h>
#include <string.h>

#define TRUE  1
#define FALSE 0

#define XSTORE(SIZE, DIVIDER, DATA, X, Y, Z ) asm __volatile__ (".byte 0xF3,
0x0F, 0xA7, 0xC0\n" \
     : "=c" (X), "=a" (Y), "=D" (Z) \
     : "c" (SIZE), "d" (DIVIDER), "D" (DATA));

#define SHA_1(SIZE, DATA, HASH, X, Y) \
       asm __volatile__ (".byte 0xF3, 0x0F, 0xA6, 0xC8\n" \
       : "=a" (X), "=c" (Y), "=D" (HASH) \
       : "c" (SIZE), "a" (0), "S" (DATA), "D" (HASH));


/* Command line parameters */

/* -bias <N>  Set the DC BIAS to the specified value [Requires wrmsr driver] */
  int dc_bias = 0;

/* -edx <N> Value of the EDX divider */
  int divider = 3;

/* -limit <X> Specify a limit at which to stop a biastest. Default 0.001 */
  float limit = 0.001;

/* -mode  Select which RNG device to use. [Requires wrmsr driver] */
  int rng_mode = 0;

/* -o     Output dir for TXT and RNG and other report files */
  char * odir = "/rngdata";

/* -pvalue <X>   Adjust chi-square pvalue to trigger full tests. Default 0.01 */
  double trigger = 0.0;

/* -raw     Evaluate raw bits [Requires wrmsr driver] */
  int raw_bits = FALSE;

/* -rep <N> Repeat N times */
  int repeat = 1;

/* -save  Save the RNG sample to disk */
  int out_flag = FALSE;

/* -sha <N> Pass bits thru SHA-1 hash. <N> bytes reduced to 20 */
  int sha_flag = 0;

/* -size <N>  Size RNG data sample - in megabytes.  Default: 10MB */
  int data_size = 0;
  int megabytes = 0;

/* -speed Just collect bits as fast as possible.  No tests */
  int speed_path = FALSE;

/* -test <N>  Tests each dc bias until or test limit [Requires wrmsr driver] */
  int biastest = 0;


/* Other globals */
```

```
   char * hostname = NULL;
   char * this_date = NULL;
   char * chip_ID = NULL;

   char flags[20] = "";
   char outfile[100] = "";

   float rate = 0.0;
   unsigned char * rng_buffer = NULL;
   unsigned int * sha_buffer = NULL;

   double ks_result = 0.0;
   double chi_result = 0.0;

   static FILE *random_out = NULL;

   int iter;
   int lines_written = 0;
   int do_bias_set = FALSE;



int acmp(const void *a, const void *b) {

   if (* (double *)a < *(double *)b)
     return -1;
   if (* (double *)a > *(double *)b)
     return 1;
   return 0;
}

void asort(double *list, int n) {
   qsort(list, n, sizeof(double), acmp);
}

double kstest(double *y, int n) {
   double t, z;
   int i;

   asort(y, n);
   z = 0.0 - (double) n * n;
   for (i = 0; i < n; ++i) {
     t = y[i] * (1.0 - y[n-1 - i]);
     if (t < 1e-20) {
        t = 1e-20;
     }
     z -= (i + i + 1) * log(t);
   }
   z /= n;

   if (z < 0.01) {
     return 0.0;
   }

   if (z <= 2.0) {
     return exp(-1.2337 / z) * 2.0 * (z / 8.0 + 1.0 - z * 0.04958 * z / (z +
1.325)) / sqrt(z);
   }

   if (z <= 4.0) {
     return 1.0 - exp(z * -1.091638) * 0.6621361 - exp(z * -2.005138) * 0.95059;
   }
```

```
  return 1.0 - exp(z * -1.050321) * 0.4938691 - exp(z * -1.527198) * 0.5946335;
}

int cleanup() {

  if (random_out)
    fclose(random_out);

  if (rng_buffer)
    free(rng_buffer);

  if (sha_buffer)
    free(sha_buffer);
}

void collect_and_hash() {

  int i,j;
  int T1, T2, T3;
  unsigned char temp_buffer[128];
  unsigned char * sha = (void *) sha_buffer;

  for (j = 0; j < data_size; j += 20) {

    sha_buffer[0] = 0x67452391;
    sha_buffer[1] = 0xefcdab89;
    sha_buffer[2] = 0x98badcfe;
    sha_buffer[3] = 0x10325476;
    sha_buffer[4] = 0xc3d2e1f0;

    XSTORE(sha_flag, divider, temp_buffer, T1, T2, T3);
    SHA_1(sha_flag, temp_buffer, sha_buffer, T1, T2);

    for (i=0; i<20; i++)
      rng_buffer[j+i] = sha[i];
  }
}


int collect_rng() {
  int T1, T2, T3;
  clock_t start = clock();

  if (sha_flag)
    collect_and_hash();
  else
    XSTORE(data_size, divider, rng_buffer, T1, T2, T3);

  return (int) ( clock() - start );
}


void processor_abort() {
  printf ("PROCESSOR ERROR.  CANNOT GET HW RNG FROM THIS COMPUTER.\n");
  cleanup();
  exit (1);
}


void validate_processor() {
```

```
/* Verify that we have the RNG and that it is enabled */
  asm ("pushl %ebx\n");
  asm ("pushl %ecx\n");
  asm ("pushl %edx\n");

  asm ("movl $0xC0000000, %eax\n");
  asm ("cpuid\n");

  asm ("cmpl $0xC0000000, %eax\n");
  asm ("jc processor_abort");

  asm ("movl $0xC0000001, %eax\n");
  asm ("cpuid\n");

  asm ("andl $0xC, %edx\n");
  asm ("cmpl $0xC, %edx\n");
  asm ("jnz processor_abort\n");

  asm ("popl %edx\n");
  asm ("popl %ecx\n");
  asm ("popl %ebx\n");
}


void process_command_line(int argc, char **argv) {

int i, file_tests;
char * c;

  for (i = 1; i < argc; i++)  {

    c = argv[i];

/* Set specific dc_bias [Requires ring 0 access/driver] */
    if (!strcmp(c,"-bias")) {
      dc_bias = atoi(argv[++i]);
      do_bias_set = TRUE;
    }

/* Specify the EDX divider value.  Default = 3 */
    if (!strcmp(c,"-edx"))
      divider = (atoi(argv[++i]) % 4);

/* Specify a biastest limit.  If the global chi-square or
   KS test approach the repective limits with this degree
   of error, terminate immediately rather than at the test
   maximum */
    if (!strcmp(c,"-limit"))
      limit = atof(argv[++i]);

/* Which set of oscillators to use [Requires ring 0 access/driver] */
    if (!strcmp(c,"-mode")) {
      rng_mode = atoi(argv[++i]);
      do_bias_set = TRUE;
    }

/* Specify directory for output reports and RNG saved data. Default /rngdata */
    if (!strcmp(c,"-o"))
      odir = argv[++i];

/* Specify a critical pvalue. If the chi-square test is less than this, save
       bits to the RNG file for later processing */
```

```
      if (!strcmp(c,"-pvalue"))
        trigger = atof(argv[++i]);

/* Use the raw bits. [Requires ring 0 access/driver] */
      if (!strcmp(c,"-raw")) {
        raw_bits = TRUE;
        do_bias_set = TRUE;
      }

/* Repeat the test(s) N times */
      if (!strcmp(c,"-rep"))
        repeat = atoi(argv[++i]);

/* Save the RNG sample to a disk file */
      if (!strcmp(c,"-save"))
        out_flag = TRUE;

/* Pass bits thru SHA-1. [Requires ESTHER processor] */
      if (!strcmp(c,"-sha"))
        sha_flag = atoi(argv[++i]);

/* Repeat the test(s) N times */
      if (!strcmp(c,"-size"))
        megabytes = atoi(argv[++i]);

/* Special speed path for power consumption testing */
      if (!strcmp(c,"-speed"))
        speed_path = TRUE;

/* Run tests on all bias settings */
      if (!strcmp(c,"-test")) {
        biastest = atoi(argv[++i]);
        do_bias_set = TRUE;
      }
    }

/* Validation and consistency checks */
    if (sha_flag > 128)
      sha_flag = 128;

    if (!do_bias_set)
      biastest = 0;

    if (sha_flag)
      strcat(flags, "H");

    if (raw_bits)
      strcat(flags, "R");

    if (biastest)
      strcat(flags, "T");

    if (repeat < 1)
      repeat = 1;

    if (limit > 0.01)
      limit = 0.01;

    if (biastest > 1000)
      biastest = 1000;

/* BIAS values are allowed only in the range 0-7. */
```

```
  if ( (dc_bias < 0) || (dc_bias > 7) )
    dc_bias = 0;

/* Make sure that we have a valid data_size for the test sequence.  In MB */
  if (megabytes < 1)
    megabytes = 10;

/* Convert to absolute number of bytes */
  data_size = megabytes * 1024 * 1024;

}



/*
   Tweaking the RNG MSR requires access to a priveleged instruction. We have
   a device driver that supports commnand-line rdmsr/wrmsr instuctions.  Any
   way you want to do this is fine, just remember that the command-line
   parameters
     -bias
     -raw
     -mode
     -test
   will cause the program to access this routine where writing the RNG MSR is
   necessary for correct program operation
*/

int c5xl_bias() {

  char cmd[60];
  int  temp;

  if (do_bias_set) {
    temp = 0x40 + (rng_mode<<8) + (dc_bias<<10) + (raw_bits<<13);
    sprintf(cmd,"wrmsr 0x110B 0x00000000%08X > /dev/null", temp);
    system (cmd);
  }
}


int datarate(float raw) {
  rate = raw / (8. * (float) data_size);
  rate = (float) CLOCKS_PER_SEC / rate;
}


#define log2of10 3.32192809488736234787


double chi_pvalue (double chi_sq, int N) {

  double p,t,a;
  int k;

  if ( chi_sq > 410. ) {
    return 1.0;
  }
  else {
    p = exp(-0.5 * chi_sq);
    k = N;

    p *= sqrt(2. * chi_sq / 3.14159265);
```

```
      while (k > 2) {
        p *= chi_sq / (double) k;
        k -= 2;
      }

      t = p;
      a = (double) N;

      while ( t > 0.0000000001 * p ) {
        a = a + 2.;
        t = t * chi_sq / a;
        p += t;
      }
      return p;
    }
}


   static long long global_counts[256] = {0};
   static long long full_size = 0;

   double * chivalues;
   static int chivalue_count = 0;


double chi_square_tests() {

   int i, k;
   int chi_counts[256];

   double prob[256];
   double expect, x, chisq;
   double ent;
   double p,t,a;


   full_size += data_size;

   for (k=0; k<256; k++)
     chi_counts[k] = 0;

   for (k = 0; k < data_size; k++)
     chi_counts[rng_buffer[k]]++;

   chisq = 0.0;
   ent = 0.0;

   expect = (double) data_size / 256.0;

   for (i=0; i<256; i++) {
     x = (double) chi_counts[i] - expect;
     chisq += x * x / expect;
     global_counts[i] += chi_counts[i];
   }

   p = 1.0 - chi_pvalue(chisq, 255);
   chivalues[iter-1] = p;

   printf("\t\tChi-square: %7.2f    PVALUE: %.5f\n", chisq, p);

   chisq = 0.0;
```

```
  expect = (double) full_size / 256.0;

  for (i=0; i<256; i++) {
    x = (double) global_counts[i] - expect;
    chisq += x * x / expect;
  }

  chi_result = 1.0 - chi_pvalue(chisq, 255);
  printf("\t\t  [Total]   %7.2f            %.5f\n",chisq,chi_result);

  ks_result = kstest(chivalues, iter);
  printf("\t\t                       KSTEST: %.5f\n", ks_result);

  lines_written += 6;

  return p;
}


int print_header() {
  printf("\tCENTAUR RNG ANALYSIS PACKAGE          %s\n", this_date);
}


int print_details() {

  printf("\t%d MB samples from %s\n", megabytes, hostname);
  printf("\tDCBIAS=%d   EDX DIVIDER=%d", dc_bias, divider);

  switch (rng_mode) {

    case 0:
    printf("  DEV=A");
    break;
    case 1:
    printf("  DEV=B");
    break;
    default:
    printf("  DEV=AB");
  }

  if (raw_bits)
    printf("  RAW BITS");

  if (sha_flag)
    printf("  SHA-1=%d", sha_flag);

  if (random_out)
    printf("\n\tRNG data saved to %s", outfile);

  printf("\n\t-------------------------------------------------\n");
}


int print_full_header() {
  print_header();
  print_details();
}


int formfeed() {
  printf("\f\n");
```

```
   lines_written = 0;
}


/* Global data needs to be reset when we switch bias settings */
int reset_vars() {

   int i;

   full_size = 0;
   chivalue_count = 0;

   for (i=0; i<256; i++)
     global_counts[i] = 0;
}


int rng_tests() {

   int k;
   int elapsed;

   time_t now;
   double p;

   now = time(NULL);
   this_date = ctime(&now);

/* Get the RNG and time the operation */
   elapsed = collect_rng();

/* Write RNG bits to disk here if requested. */
   if (out_flag)
     fwrite(rng_buffer, 1, data_size, random_out);

   datarate(elapsed);

   if (lines_written == 0)
     print_full_header();

   printf("\n\tSample %d.  RNG datarate: %d kbits/sec\n", iter, (int)
rate/1000);

   if (speed_path)
     return 0;

   p = chi_square_tests();

   printf("\t\t----------------\n");

/* If the user has requested a critical p-value save the data now.
    The command-line parser makes sure that if out_flag == TRUE
    then trigger value = 0.0 */
   if ( p < trigger )
     fwrite(rng_buffer, 1, data_size, random_out);

   if (lines_written > 45)
     formfeed();

}
```

```
int main (int argc, char **argv) {

   time_t now;
   char s[12];
   char report_file[50];
   char * temp;
   int i;
   int mtbf[8] = {0};

   validate_processor();
   chivalues = (double *) valloc(1000000 * sizeof(double));

   process_command_line(argc, argv);

   rng_buffer = valloc(data_size + 8);
   sha_buffer = valloc(128);



   if (!rng_buffer || !sha_buffer) {
      fprintf(stdout,"Unable to allocate buffer for RNG\n\n");
      cleanup();
      exit (1);
   }

   hostname = getenv("HOSTNAME");

/* Report file should be DATE.FFFFFF.txt
    where the F's are flags to indicate which tests
    are reported and test parameters */

   now = time(NULL);
   strftime (s, 12, "%y%j.%H%M", localtime(&now));

   sprintf(report_file, "%s/%s.%s.%d%s.TXT", odir, chip_ID, s, divider, flags);
   sprintf(outfile,     "%s/%s.%s.%d%s.RNG", odir, chip_ID, s, divider, flags);

/* Can we write the requested output file?? */
   if (out_flag || (trigger > 0.0) ) {

      random_out = fopen(outfile, "wb");

      if (!random_out) {
         fprintf(stderr, "Error creating RNG outfile %s\n\n", outfile);
         cleanup();
         exit (1);
      }
   }

   if ( !biastest ) {
      c5xl_bias();

      for (iter = 1; iter <= repeat; iter++)
         rng_tests();
   }
   else {

      for(rng_mode = 0; rng_mode < 3; rng_mode++) {

         for (i = 1; i <= repeat; i++) {
            for (dc_bias=0; dc_bias<8; dc_bias++) {
```

```
            c5xl_bias();

            for (iter = 1; iter <= biastest; iter++) {
              rng_tests();

              if ((iter>3) && ( (ks_result>(1.0-limit)) || (chi_result<limit)))
                break;
            }

            mtbf[dc_bias] += iter;
            formfeed();
            reset_vars();
          }

          print_header();

          printf("\n\t--------------------------------------\n");
          printf("\n\n\n");

          for (dc_bias = 0; dc_bias < 8; dc_bias++)
            printf("\tMTBF DC BIAS = %d:  %d\n",dc_bias, mtbf[dc_bias] / i );

          formfeed();
        }

        for (dc_bias = 0; dc_bias < 8; dc_bias++)
        mtbf[dc_bias] = 0;
      }
    }

    cleanup();
    exit(0);
}
```